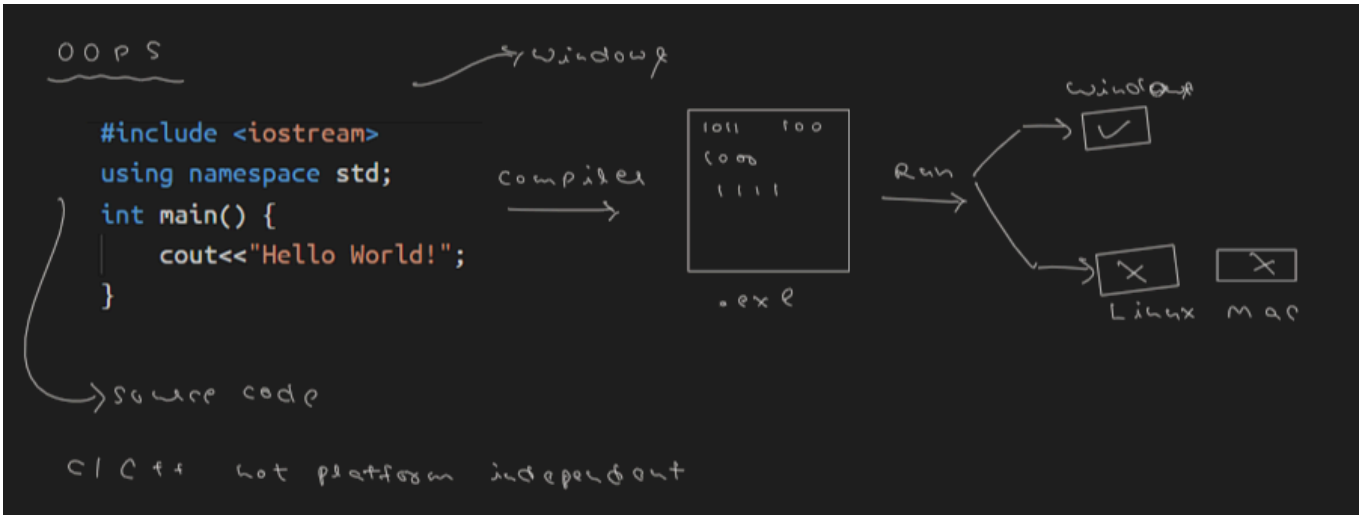
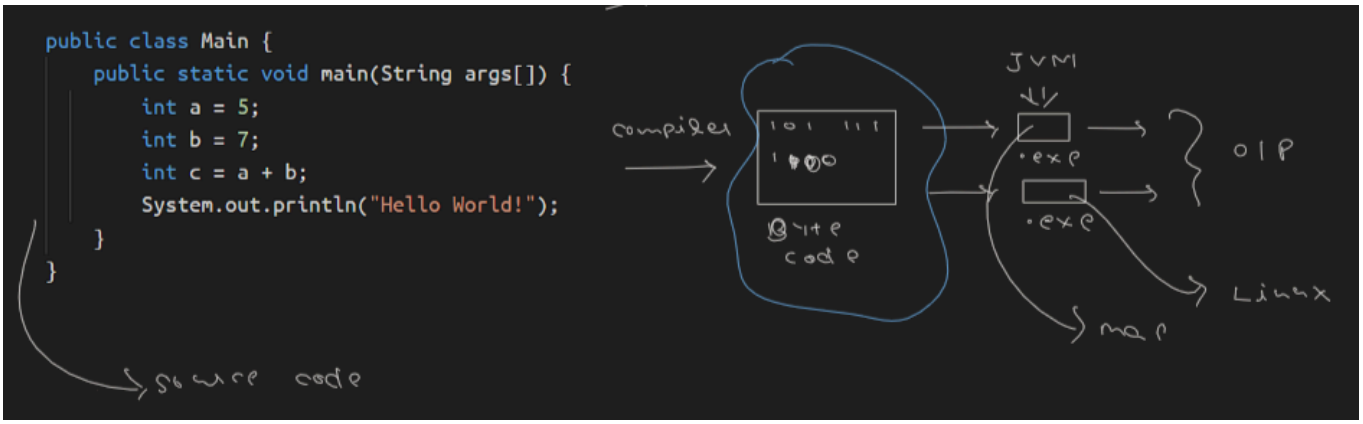
💻 C++ is Not Platform Independent

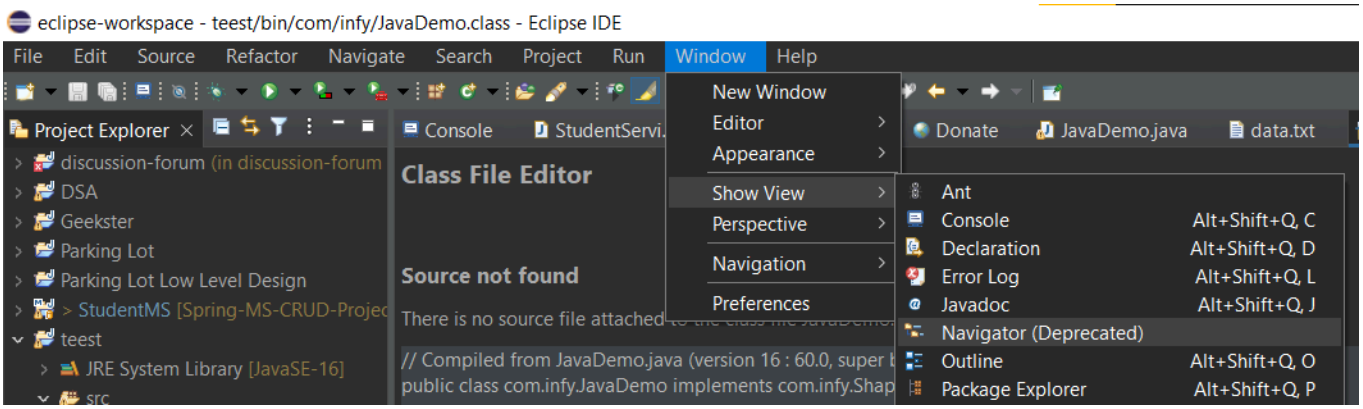


- ➔ C++ is not a platform-independent language. Suppose I have written a piece of code on a Windows machine. When I compile it, it generates an `.exe` file which contains binary values.
- ➔ Now, when I click the **Run** button, it shows the output. But when I try to run that `.exe` file, it will only work on a **Windows** machine, because only Windows understands the format in which the program was compiled.
- ➔ It will **not run** on a **Mac** or **Linux** machine, because the source code was compiled on Windows. So, the `.exe` will only work on the Windows platform.
- ➔ If you send the `.exe` file, it will not work on other platforms. But if you send the `.cpp` file, it will work on all platforms.

☕ Why Java is Slower but Platform Independent

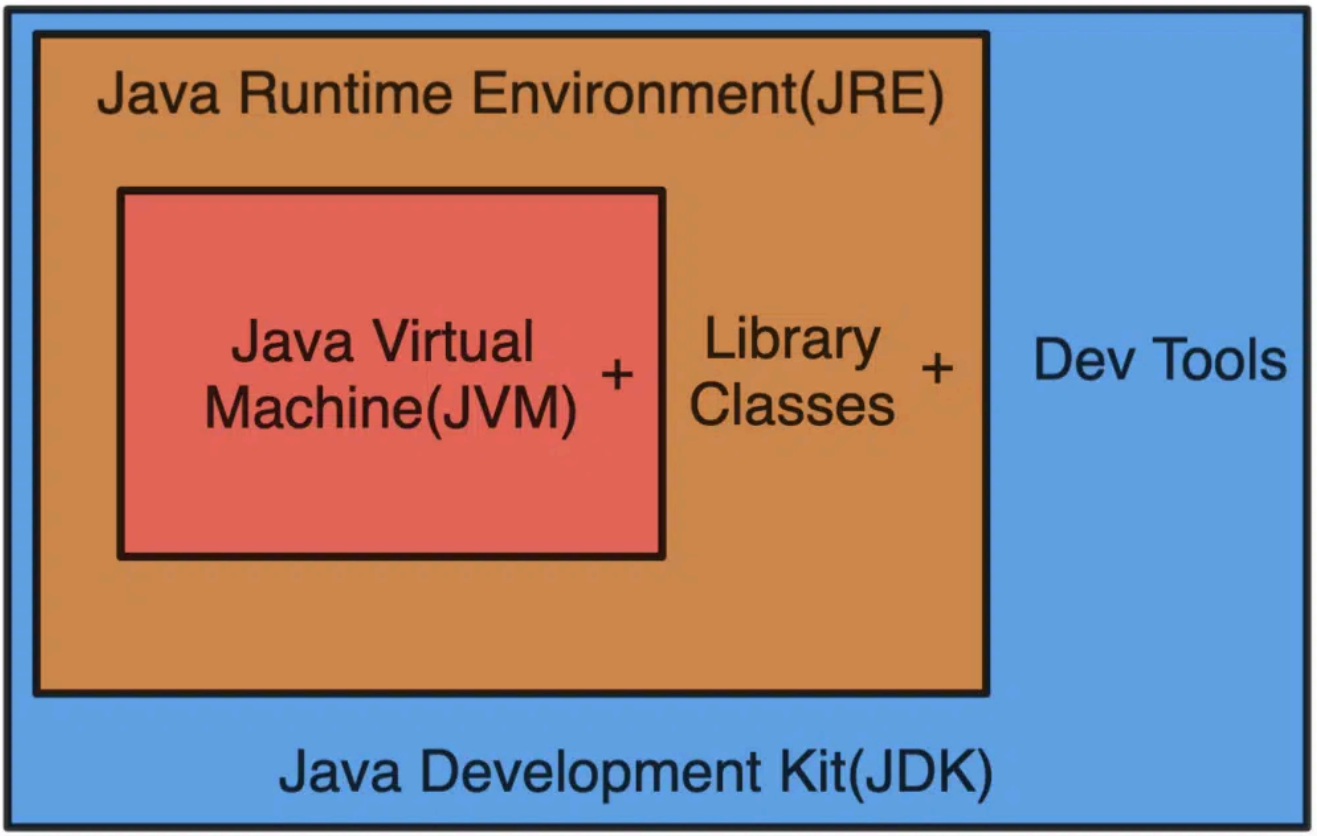


- ➔ Java is **platform-independent** but generally **slower** compared to languages like C++.
- ➔ Suppose I have written a piece of Java code on a **Windows** machine. When I compile that code, a `.class` file is generated. This `.class` file contains **bytecode** — an intermediate form of the program — which is usually stored in the `bin` folder, while the original Java code is stored in the `src` folder.
- ➔ Now, if you supply this `.class` file to **Windows**, **macOS**, or **Ubuntu**, the **JVM (Java Virtual Machine)** on each platform will execute it. This is what makes Java **platform-independent** — the same bytecode can run on any system that has a compatible JVM.
- ➔ However, because of this **extra step** (executing through the JVM instead of directly running machine code), Java programs tend to be **slower** compared to compiled languages like C++, which generate platform-specific executables.



☕ JVM, JRE, and JDK – What You Need to Run or Compile Java

- ➔ If you just have the **class file**, the **JVM** will execute the file.
  If you have some libraries also in your code, then the whole **JRE** will be needed.
- ➔ Now, if someone has provided you a **Java file**, then to execute it you need the **compiler** also — basically the **dev tools**.
  The whole thing is known as the **JDK**.



🔄 Why Object-Oriented Programming?

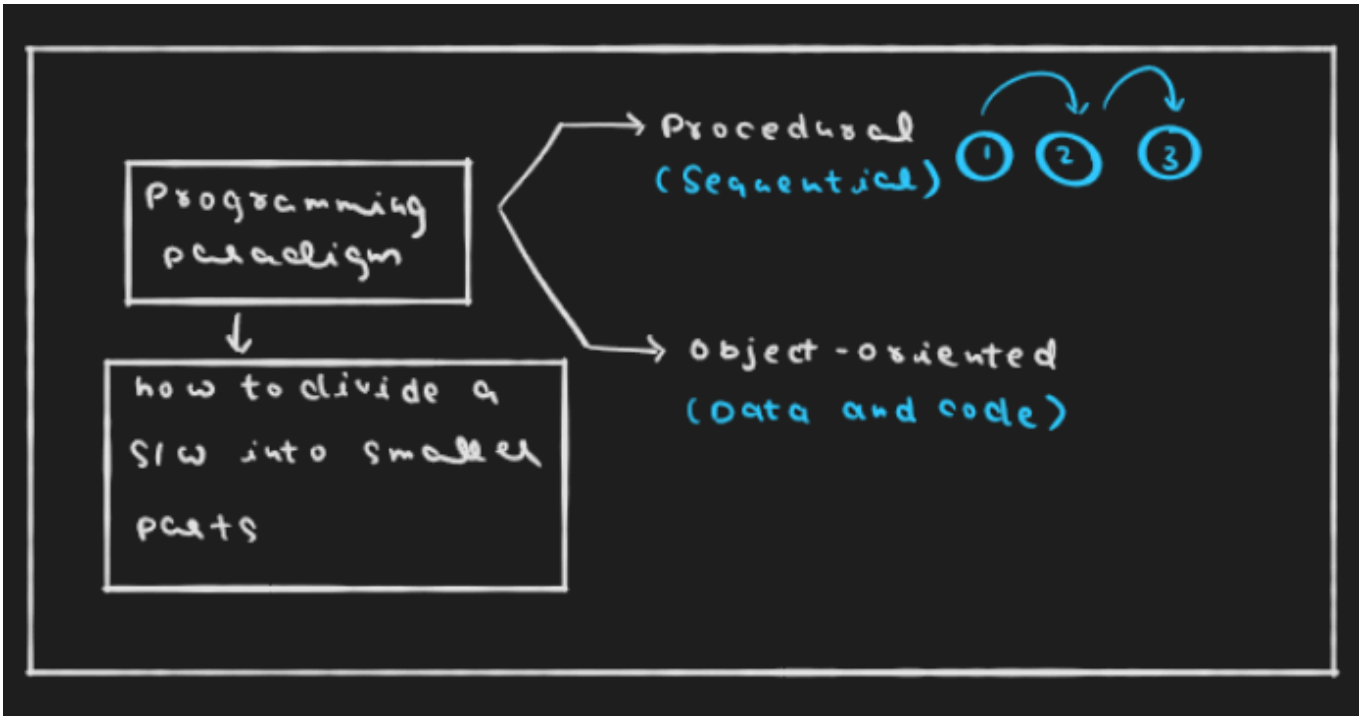
See, whatever code we have written till now was **function-oriented**, like we used to write in **C**. That kind of code is not reusable and not everyone can use it easily.

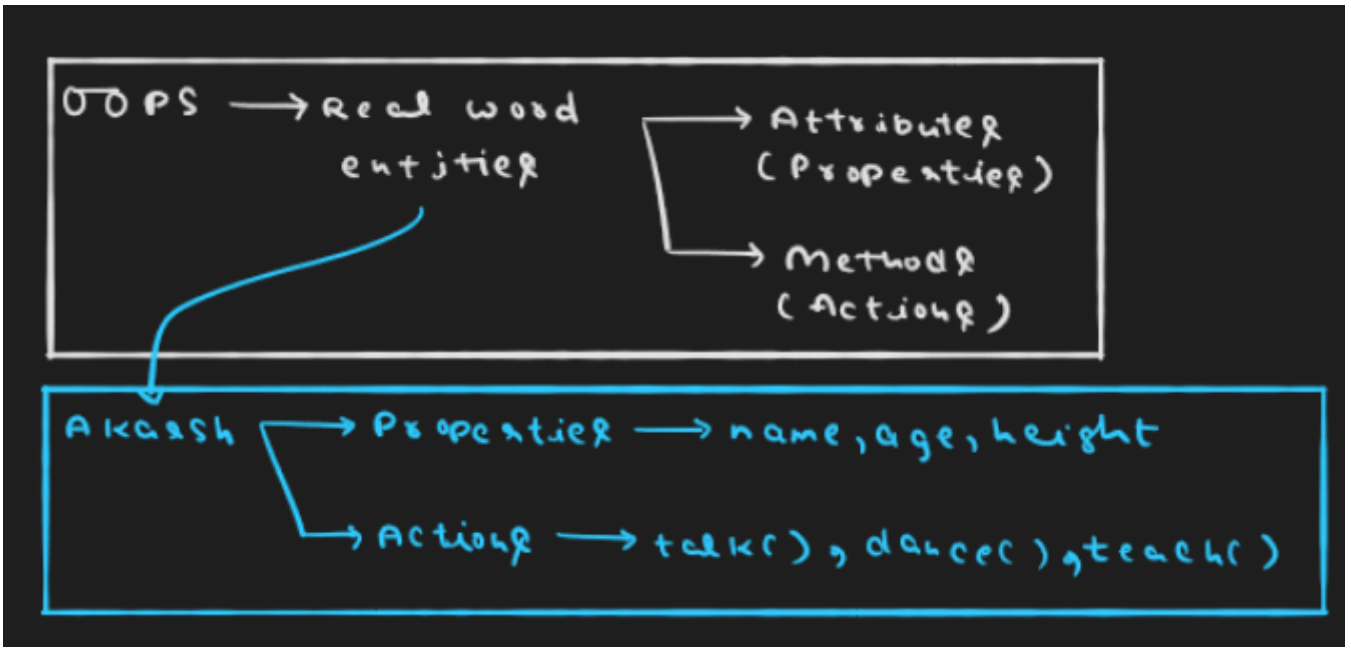
But **Java** has given us a list of **objects** that we can utilize very easily.

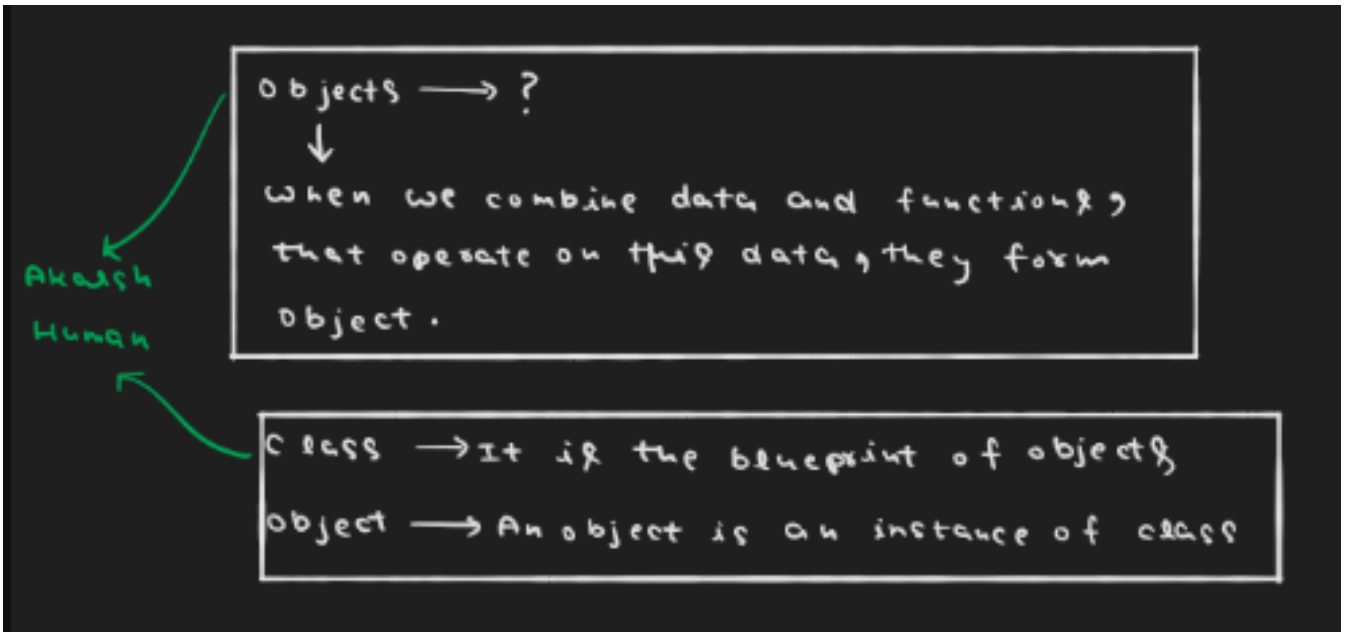
So, we want to write code that:

- Everyone can use
- Is easy to use
- Is easy to maintain
- Is reusable and modular

That's why we move towards **Object-Oriented Programming** — to make our code clean, structured, and reusable.


Programming paradigm
how to divide a s/w into smaller parts
Procedural (Sequential)
1
2
3
Object-oriented (Data and code)


OOPS → Real word entities
Attributes (Properties)
Methods (Actions)
Akash
Properties → name, age, height
Actions → talk(), dance(), teach()


Objects → ?
When we combine data and functions, that operate on this data, they form object.
Akash
Human
Class → It is the blueprint of objects
object → An object is an instance of class

🔁 Reusability in Java Using Object-Oriented Programming

```
ArrayList<Integer> ll = new ArrayList<>();
ArrayList<Integer> ll1 = new ArrayList<>();
```

In the above example, if you notice, there is **no `main` method** and no **`public static` method** in the declaration itself. Still, we are able to **reuse the functionality** with two different data sets.

For example, `ll` is one list and `ll1` is an entirely different list — there is **nothing common** between them in terms of data; they are **independent**.

Yet, we are using the **same functionality**, which shows how reusability works in Java through object-oriented programming.

🎯 Student Example

```java
public class Student {

   String name;
   int age;

   public void introYourSelf() {
       System.out.println("My Name is " + this.name + " & age is " +
this.age);

   }

   public void sayHey(String name) {
       System.out.println(this.name + " says Hey " + name);
       introYourSelf();
   }

   public static void mentorName(Student s) {
       // cannot access non-static part in static area
       // introYourSelf(); // error
       System.out.println("Akarsh is mentor of " + s.name);
   }

   static {
       // static block
       System.out.println("I am in Student class 1");
   }

   static {
       // static block
       System.out.println("I am in Student class 2");
   }

}
```


```java
public class StudentClient {
    public static void main(String[] args) {
//      ArrayList<Integer> ll = new ArrayList<>();
//      ArrayList<Integer> ll1 = new ArrayList<>();

        System.out.println("Hello Akarsh");

        Student s = new Student();
        System.out.println(s.name);
        System.out.println(s.age);

        s.name = "Shinchan";
        s.age = 10;
        s.introYourSelf();

        Student s1 = new Student();
        s1.name = "Nohara";
        s1.age = 21;
        System.out.println(s.name);
        System.out.println(s.age);
        s1.introYourSelf();

        s.sayHey("Gautam");

        Student.mentorName(s);
        Student.mentorName(s1);
    }

    static {
        // static block
        System.out.println("I am in StudentClient  class 1");
    }

}
```

📚 Why No `main` Method in the Student Class?

- ➔ I have not written the `main` method in the `Student` class because I want to make it act like a **library** that anyone else can use — just like the **List** library that Java provides.
- ➔ That's why I will call it from **another file or class**, where the `main` method exists. That class will create an object of the `Student` class and use its functionality.

🔑 Understanding the `this` Keyword

- ➔ The `this` keyword is used to refer to the **current class members**.
- ➔ For example, if I have a variable called `name` in the current class, and I am also passing a parameter with the same name, then using `this.name` will refer to the **class-level variable**, not the parameter.

🧠 How `this` Stores Object Reference

- ➔ Whenever you call a function — suppose from the object `s` — the **address of that object** is also passed internally.
- ➔ That address is stored in the **`this` keyword**, and it can be used to **access the current object** and to **identify which object has called the method**.

⚡ Role of Static Block in Java

- ➔ A **static block** will always get executed **at the starting even before the main method will get executed.**

🏷️ Static Methods and Variables Are Shared

- ➔ **Static method/variable** is **common for all objects** and **referenced using class name**, like mentorName method is static which will remain same for any student.

🚫 Access Rules: Static vs Non-Static

- ➔ From a **static context**, you **cannot access** a non-static member directly, and **vice versa**.
- ➔ You can only:
  - ◆ Access a **static part from another static part**
  - ◆ Access a **non-static part from another non-static part**

🔹 JVM and main()

The **JVM (Java Virtual Machine)** always starts executing a Java program from the **main function**. When the JVM loads a class to run, it looks for the **main method as the entry point**. There must be only one **main method with the correct signature** in a class—only a single instance of it can exist.

The JVM begins execution from this method and continues running the program from there. Also, the Java program is executed by the **JVM**, which runs the program from **outside the class**.

For the JVM to access the main method, it must be declared as **public**. If it's marked **private**, the JVM won't be able to see or access it, because private methods are only visible within the class. Declaring it **public** ensures that the JVM can **invoke it from outside the class**.

🎯 Person Example

```
public class Person {
   String name = "Akarsh";
   int age = 25;

   //default constructor by Java (only when no human constructor is formed)
```

```
    public Person() {
    }

    public Person(String name ,int age) {
        System.out.println("constructor");
        this.name=name;
        this.age=age;

    }
//    public Person(String name ) {
//
//    }
//    public Person(int  age ) {
//
//    }

  static {
     System.out.println("Hey static");
  }

}
```

```
public class PersonClient {
    public static void main(String[] args) {
        System.out.println("Hi Akarsh!");

        Person p = new Person("Shinchan", 21);
        p.age = 42;
        p.name = "Shinchan Nohara";
//      p.age=22;

        Person p1 = new Person("Doaremon", -20);
    }

}
```

🛠️ Understanding Constructors and Polymorphism in Java

A **constructor** is used to **initialize the attributes** of a class.

Now, whenever you **instantiate a class**, first the **static block** is executed, and then the **constructor** is called **automatically**.

Constructors can be of multiple types:

- **Default constructor** (without parameters)
- **Parameterized constructor** (with one or more parameters)

You can also define **multiple constructors** in the same class — this is what **polymorphism** is all about: the ability to exist in **multiple forms**.

If you look at an example, you might see:

- A constructor with **one parameter**

- Another with **two or three parameters**
- And one with **no parameters**

When you create an object and pass a specific number of arguments, the **matching constructor** is called automatically.

⚙️ How a Java Program Executes (Step-by-Step)

First, **memory allocation** takes place — the variables like `name` and `age` are declared and assigned default values (e.g., `null` for objects, `0` for integers).

In the **second step**, **parsing** happens — this means whatever **default values** you've provided while creating the class (like during variable declaration) will be assigned.

In the **third step**, the **constructor** is called, and whatever values are passed into the constructor will be used to **initialize the variables**.